Training Hierarchical Networks for Function Approximation

by

Brando Miranda
B.S., C.S. M.I.T., 2014

Submitted to the Dept. of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

© Massachusetts Institute of Technology 2016. All rights reserved.

Author ............................................................

Dept. of Electrical Engineering and Computer Science
September 9, 2016

Certified by .........................................................

Tomaso Poggio
Eugene McDermott Professor in the Brain Sciences
Thesis Supervisor

Accepted by .........................................................

Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee
Training Hierarchical Networks for Function Approximation

by

Brando Miranda

Submitted to the Dept. of Electrical Engineering and Computer Science on September 9, 2016, in partial fulfillment of the requirements for the degree of Master of Engineering

Abstract

In this work we investigate function approximation using Hierarchical Networks. We start of by investigating the theory proposed by Poggio et al [2] that Deep Learning Convolutional Neural Networks (DCN) can be equivalent to hierarchical kernel machines with the Radial Basis Functions (RBF). We investigate the difficulty of training RBF networks with stochastic gradient descent (SGD) and hierarchical RBF. We discovered that training singled layered RBF networks can be quite simple with a good initialization and good choice of standard deviation for the Gaussian. Training hierarchical RBFs remains as an open question, however, we clearly identified the issue surrounding training hierarchical RBFs and potential methods to resolve this. We also compare standard DCN networks to hierarchical Radial Basis Functions in tasks that has not been explored yet; the role of depth in learning compositional functions.

Thesis Supervisor: Tomaso Poggio
Title: Eugene McDermott Professor in the Brain Sciences
Acknowledgments

I would like to thank my advisor Tomaso Poggio for his support throughout this project. Tommy trusted me to tackle the problem in my own way, provided me with guidance and connected me with all the people who made this possible.

I would also like to thank Gemma Roig for her patience and suggestions in all the discussion we had for making this thesis possible.

The most challenging part of a project can be the beginning. Thus, I’d like to thank Fabio Anselmi for guiding me during the first steps in our project and making this thesis possible.

I would also like to thank my very supportive team in the Poggio Lab at MIT.

I would also like to thank my long time friend Pedro Cattori for providing me with advice when it came to software design.

Finally, I thank my mother Beatriz Garay, my father Brando Miranda, and my girlfriend, Yasmin Chau, for their committed support at every moment during my research and life.
## Contents

1 Introduction 7

2 Regression Functions 9
   2.1 Introduction ......................................................... 9
   2.2 2-Dimensional regression functions $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ .................. 10
      2.2.1 Quadratic plus sine curve ($h_{add}$) .............................. 10
      2.2.2 Training HBF .................................................... 10
      2.2.3 Reconstruction of $h_{add}$ ...................................... 12
      2.2.4 Reconstruction of $h_{add}$ with more centers .................... 15
   2.3 Gabor function ($h_{gabor}$) ........................................ 19
      2.3.1 Training HBF .................................................... 19
      2.3.2 Motivation for training the standard deviation ................. 21
      2.3.3 Reconstructions $h_{gabor}$ ..................................... 22

3 Auto-encoders 25
   3.1 Introduction ......................................................... 25
   3.2 Different types of initializations for the non-linear models ........ 25
   3.3 Comparing the neural networks and Hyper Basis Functions (HBF) . 28

4 The Role of Structure in Learning Compositional Functions 30
   4.1 Introduction ......................................................... 30
   4.2 Experimental setup .................................................. 30
   4.3 Learning a 1 dimensional compositional function .................... 31
4.3.1 Hierarchical neural networks ........................................ 31
4.3.2 Hierarchical Hyper Basis Function networks .................. 32
4.3.3 Tricks for training the hierarchical HBF for \( f(x) = 2(2\cos^2(x) - 1)^2 - 1 \) .......................................................... 33
4.3.4 Discussion .............................................................. 35
4.4 Learning a 2 dimensional compositional function .................. 35
4.4.1 Neural Network training ............................................. 35
4.5 Learning MNIST with Hierarchical Models .......................... 36
4.6 Learning binary tree function with CNNs ............................ 37
4.6.1 Network structures .................................................. 38
4.6.2 Result of learning 4D binary tree function ....................... 38
4.6.3 Discussion for 4D function ......................................... 39
4.7 Learning an 8D binary tree function .................................. 39
4.7.1 Network structures .................................................. 40
4.7.2 Results of learning 8D binary tree function ....................... 41
4.7.3 Discussion for 8D function ......................................... 42

5 Barrier to Training Hierarchical HBFs ................................. 43
5.1 The trade-off that results in permanent vanishing gradients ......... 43
5.1.1 Unstable pre-synaptic activation .................................. 44

6 Applying Batch Normalization to Hierarchical HBFs ................. 45
6.1 Background ............................................................. 45
6.2 Naive batch normalization on hierarchical HBF ..................... 46
6.3 Suggestion 1: applying batch normalization to Euclidian distance before squaring ................................................. 47
6.4 Suggestion 2: apply batch normalization to squared Euclidian distance and square it again ............................................. 48
6.5 Suggestions to fix the optimization procedure for hierarchical HBFs 49
6.6 Normalize filters and activations of the Gaussian ..................... 49
6.7 Use the Laplacian function instead of Gaussian ..................... 50
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.7.1</td>
<td>Suggestion on applying Batch Norm to Laplacian pre-activation</td>
<td>51</td>
</tr>
<tr>
<td>7</td>
<td>Further Work</td>
<td>53</td>
</tr>
<tr>
<td>7.1</td>
<td>Memory minimization</td>
<td>53</td>
</tr>
<tr>
<td>7.2</td>
<td>Removing internal covariate shift for hierarchical HBFs</td>
<td>53</td>
</tr>
<tr>
<td>7.3</td>
<td>Removing the trade off for signal propagation in hierarchical HBFs</td>
<td>54</td>
</tr>
<tr>
<td>7.4</td>
<td>Using HBFs to beat other machine learning algorithms on function approximation</td>
<td>54</td>
</tr>
<tr>
<td>7.5</td>
<td>Regularization</td>
<td>54</td>
</tr>
<tr>
<td>8</td>
<td>Appendix</td>
<td>55</td>
</tr>
<tr>
<td>8.1</td>
<td>Update equations using back-propagation for layered HBF network</td>
<td>55</td>
</tr>
<tr>
<td>8.2</td>
<td>Equations for derivatives</td>
<td>56</td>
</tr>
<tr>
<td>8.3</td>
<td>Computing delta</td>
<td>56</td>
</tr>
<tr>
<td>8.4</td>
<td>GPUs</td>
<td>57</td>
</tr>
<tr>
<td>8.5</td>
<td>Software</td>
<td>57</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The architectures now called Deep Learning Neural Networks (DCN) have achieved impressive performances in vision and speech classification tasks. The performance of these systems is ironically matched by our present lack of understanding of why these systems work as well as they do. In the aspirations to bring new insights to why Neural Networks work, we propose to do empirical experiments on an potentially equivalent model called hierarchical HyperBasis function networks and explore their performance in function approximation.

As it was shown by Poggio et al. [2] DCNs are hierarchies of kernel machines. In particular, with normalized inputs, it is possible to show [2] that DCNs are hierarchies of radial kernel machines, also called Radial Basis Functions (RBFs). In particular the non-linearity is in fact the RBF kernel. This means that each layer of DCN is a radial kernel but since the weights and templates of the RBFs are learned in a supervised way, the kernel machines should be more properly called Hyper Basis Functions (HyperBF or shortly HBFs) [10]. When the centers are the data set points then we have a Radial Basis Function (RBF) network.

In this work we wish to compare hierarchies of HBFs with the standard RBF networks and DCNs. To achieve this we will train a single layer HBF network with Stochastic Gradient Descent and compare it with the standard RBF network with fixed centers. Then we aim to show that one can train a 2 layer HBF network with stochastic gradient descent. With that we want to compare the statistical performance
of a 2-hidden layer with the single layer HBF and RBF network. Furthermore, we want to compare them to Neural Networks as well.

It is known that encoding the compositional structure of the function to be approximated in the network itself guarantees the existence of a good approximation [7]. It was shown in [7] that when the structure of the network incorporates (the prior knowledge) the compositional structure of the function being learned, then there are significant statistical performance can benefit. For example, one can show that the number of parameters one needed in theory to approximate a compositional function to the same accuracy as a shallow network is exponentially less. In this work we investigate this theory by showing the better optimum can be found by optimization procedures in a few examples functions.

Thus, learning representations from natural images and learning compositional functions are the natural tasks to investigate. Therefore, we want to compare all these different types of HBF models (singled layered and double layered) to DCNs in the task of auto-encoding and function approximation of compositional functions. We wish to compare these different algorithms on a standard data set such as the MNIST data set for recognizing handwritten digits.
Chapter 2

Regression Functions

2.1 Introduction

In this section we will study the optimization of singled layered HBF networks via SGD and its performance on function approximation. When doing function approximation we will be comparing the Nystrom subsampled Kernel regression [11] against the same network but additionally training the centers $t$ in $k(\cdot, t)$. The maximum number of centers that we will train is about $O(\sqrt{N})$ since this is the number were generalization can be guaranteed in the Nystrom subsampled setting. Of course it will be important to check if generalization improves further when training the centers. Its important to emphasize that training the centers does not make sense when we do not subsample.

Throughout this document we will refer to the kernel with moving centers as HBF and the one with stationary centers as RBF. For a singled layered HBF network will be abbreviated as HBF1, two layered HBF network HBF2, and son on.
2.2 2-Dimensional regression functions $f : R^2 \rightarrow R$

2.2.1 Quadratic plus sine curve ($h_{add}$)

In this subsection we introduce the function $h_{add}$ to be learned (function borrowed from the paper by Poggio, Girosi and Jones [4]):

$$h_{add}(x, y) = \sin(2\pi x) + 4(y - 0.5)^2$$

The following figure contains the original $h_{add}$ function to be approximated:

![Original function](image)

2.2.2 Training HBF

We will train an HBF model via SGD with up to $\sqrt{N} \approx 180$ number of centers. Training was done by first learning a standard RBF network and then choosing as initialization for the HBF network the best RBF network. The best RBF network was chosen among different initializations via standard hold-out validation. In this procedure we subsampled the training set randomly for the initial centers and chose
the one with the smallest validation error.

One can visually observe that for the function $h_{add}$ a standard deviation of about 2 units is a reasonable choice. However, for a more robust method that generalizes better to different data sets, it’s better to choose the standard deviation of the RBF via cross-validation, grid search or random search as suggested by Bergstra and Bengio [3].
Figure 2-1: Comparison of HBF1 vs RBF with test and train set. Notice that the test error is lower for HBF1. The increase in error between 25 to 60 centers happened because of the stochasticity of the training processes.

We can see from figure 1 that training the centers decreases the train error and test error. As the number of centers is increases up to \( O(\sqrt{N}) \) we observe a decrease in training error (as expected). Note that training the centers of the subsampled model did not introduce overfitting (i.e. the test error also decreased). This can be appreciated most when with a few number of centers (i.e. when the number of centers is \( \ll O(\sqrt{N}) \)). Given a low number of centers one can gain much more (generalization) by training the centers than when having more centers. This can be appreciated by the large gap between the errors when with few centers is used. This gap closes when the number of centers approaches \( \sqrt{N} \approx 180 \).

2.2.3 Reconstruction of \( h_{add} \)

To explore the quality of our learning at a qualitatively level it is often useful to make reconstructions when possible. In this case we will appreciate the correct shape of
the function is learned. This analysis will also help us understand the advantage of moving the centers presents.

Recall the figure of the original function that our methods were trained on:

Now we will observe the reconstruction that resulted from training the HBF1: There are two figures and one has the reconstruction using HBF (moving centers) and the reconstruction using the subsampled kernel RBF:
One can appreciate that the heights of the reconstructions of the RBF seem rather random compared to the heights of the reconstructions of the HBF. This is a direct consequence of sumpsampling the kernel matrix randomly. Furthermore, once the
RBF centers are fixed the training algorithm can only learn function heights on the locations that have centers. This means that once the centers are fixed, if the centers chosen are on locations where the original function has low signal, then we cannot extract much information. This is why one needs about $\sqrt{n} \approx 180$ centers to generalize well because the more centers we choose the higher chance of landing on locations with signal. This is why its often better for generalization to have many centers from the data set (where using the full set of centers is optimal according to the representer theorem).

However, using moving centers we can depend less on choosing lucky centers. If the centers that we initially chose are not good, then one can simply train and thus move the centers so to minimize the error via learning. Intuitively, there are two type of learnings: one that got stuck with the initial centers (RBF) and a second one that was able to artificially probe for better centers via learning and stochastic gradient descent (HBF). This resulted in HBF having a generalization matching the RBF with $\sqrt{n}$ centers even though we had less centers. This was possible because we allowed the centers to move.

To say this a different way, we can read the graph by specifying a certain accuracy or error $\epsilon$. If we have a target error we can find out the number of centers each models needs. Then the HBF will require less centers and provide as good quality of approximation because it allows to change our initial bad guesses to better estimates.

Obviously, there must be a lower bound on how many centers we need even if we allow them to be trained and that can be a interesting theoretical question to pursue. I conjecture that the lower bound on the number of centers needed will be around $\Omega(\log N)$.

### 2.2.4 Reconstruction of $h_{add}$ with more centers

An interesting to notice is there the presence of diminishing returns. The more number of centers that we require, the less of an improvement there will be. This diminishing return can be appreciated by comparing the following reconstructions:
Compare the quality of reconstruction between the two figures. We increased the number of centers and there is an improvement however, the improvement is small. The closer we get to the optimal number of centers we needed the less an additional
center will improve the approximation.

Also notice that once the number of centers is high (close to $\sqrt{n}$), training the centers is provides small benefit compared to when there are few centers. One can appreciate this with the first figure provided where the error gap is larger with few centers than with more centers. To further emphasize this observe the reconstruction function when the number of centers is $\sqrt{N}$:
The reconstruction due to the HBF does provide a smoother and slightly better reconstruction but the question is if it’s worth the additional time to do gradient descent once the error is already small at this point.

In conclusion if the number of centers is large, the benefits of gradient descent start to diminish.
2.3 Gabor function \( (h_{gabor}) \)

To make our conclusion about the learning task more robust, it is important to approximate different functions. Therefore, in this section we observe how the HBF1 learns the function \( h_{gabor} \) (inspired from the paper by Poggio, Girosi and Jones\[4\]):

\[
h_{gabor}(x, y) = e^{-(x^2+y^2)} \cos(2\pi(x+y))
\]

The following figure contains the \( h_{gabor} \) function:

2.3.1 Training HBF

In this task we repeated the the way that we learned the HBF as in the last section. The HBF1 is mostly better than RBF, specially when the number of centers is small (similarly as in the previous experiments). However, \( h_{gabor} \) proved more difficult to successfully learn. The learning task for this function was much more sensitive on the choice of standard deviation. With a bad standard deviation it is very difficult
to successfully train the HBF model. This is not surprising because as the Euclidian distance between centers and data point increases, the value of the RBF exponentially decreases.

When the standard deviation was not chosen correctly there was little learning. Then the shape of the Gaussian is to small then the RBF function tends to have really small activation value. Thus, choosing the standard deviation right for the optimization of an HBF. However, when the shape of the Gaussian is large enough it is able to provide to stochastic gradient descent to make learning possible.

This means that choosing a good standard deviation from cross validation in a RBF network is crucial. Furthermore, it suggests that potentially, it can be very important to train a this parameter. This way one could potentially not only choose a good standard deviation at initialization but also SGD could (hopefully) improve our choice.
2.3.2 Motivation for training the standard deviation

The problem of choosing the standard deviation can also be better understood in a theoretical perspective by inspecting the pre-activation function $s_j^{(l)}$:

$$s_j^{(l)} = -\beta \|a^{(l-1)} - w_{i,j}^{(l)}\|^2$$

$$2\beta \langle a^{(l-1)}, w_{i,j}^{(l)} \rangle - \beta (\|a^{(l-1)}\|^2 + \|w_{i,j}^{(l)}\|^2) = 2\beta \langle a^{(l-1)}, w_{i,j}^{(l)} \rangle - \beta (B(a^{(l-1)}, w_{i,j}^{(l)}))$$

consider comparing it with the usual pre-activation of a Neural Network:

$$s_j^{(l)} = \langle a^{(l-1)}, w_{i,j}^{(l)} \rangle + b_j^{(l)}$$

(notice that $a^{(l-1)} = \theta(s_j^{(l)})$ for activation function $\theta$)

Consider fixing the offset of a Neural Network to zero. It’s not too hard to notice that then one removes the universality property of a Neural Network. Its import to allow the offset to move for it to be able to give it the chance to approximate any function.

Similarly here, if we simply do not train $\beta$ (or at least allow it to change via cross-validation) one is potentially hindering HBF model because $\beta$ is the only variable that controls the "offset" (labeled $B(a^{(l-1)}, w_{i,j}^{(l)})$) of the HBF. It is true that $w_{i,j}^{(l)}$ also controls it, but training $\beta$ provides an additional way to control it to.

Notice that by normalizing the data and the centers then we have $B(a^{(l-1)}, w_{i,j}^{(l)}) = 1$ and thus get:

$$-\beta \|a^{(l-1)} - w_{i,j}^{(l)}\|^2 = 2\beta (\langle a^{(l-1)}, w_{i,j}^{(l)} \rangle - 1) = 2\beta \langle a^{(l-1)}, w_{i,j}^{(l)} \rangle - \beta$$

which would be the only way to adapt the offset. If the data is normalized in this way, training $\beta$ is not optional but a necessity for the HBF to learn. This provides further evidence that training the standard deviation is potentially useful for improving the HBF.
2.3.3 Reconstructions $h_{gabor}$

To ease a qualitative understanding of training when the standard deviation is small it's useful to consider reconstructions.

Recall the true Gabor function:
Now consider what the RBF (subsampled kernel method) learned against what the HBF (moving center function) learned:
In this case one can notice that the HBF was better at learning the negative part of the Gabor function, and in fact, most of the reconstructions reconstructed the negative part of the functions much better.

In summary however, both methods did not do a good job in reconstructing the Gabor function and therefore showing that careful choice of the shape of the Gaussian is quintessential for effective use of the RBF and HBF model.
Chapter 3

Auto-encoders

3.1 Introduction

In this section we discuss the performance of the HBF with the task of auto encoding the MNIST data set. The data set was normalized by dividing each pixel by 255, however, no further pre-processing was done to the data.

When trying to perform auto encoding its important to have a benchmark to know how good our auto-encoder is doing. For this we will use the standard RBF model and PCA as a benchmark.

We are interested in knowing for a given compression rate (in this case for a fixed number of centers), which algorithm, HBF, RBF or a Neural Network with either a ReLu as an activation or a Sigmoid function as an activation performs the best reconstruction.

3.2 Different types of initializations for the non-linear models

For the non-linear models there were different initializations that were tried.

- Random initialization according to a Gaussian $t \sim N(0, \epsilon^2)$ (which is the standard one used for neural networks)
• Using the Kernel Model itself as an initialization (i.e. using whatever was learned from \( c = K^+\hat{Y} \) as initial weights)

• Using the random data set points as centers

• Using the final weights to be random in the range of the signal of the function (i.e. random value with average \( y \) value from the data set).

• Using \( k++ \) initialization to choose centers so to have centers with high distance between each other

• and different combinations of all the above (and others that didn’t work well or made small different)
First let’s consider training the HBF with $t \sim N(0, \epsilon^2)$ and comparing how it does with respect to a kernel method with centers as data points:

This graph reveals that this type of initialization did not work very well with this type of activation function even when one moved the centers (notice that the kernel one uses data as centers and using this centers was not good). There was not much benefit from moving the centers in this case because the initializations is not very good for the HBF. Therefore, this suggests that the method of training the centers via SGD is sensitive to initialization (as its a non-convex optimization problem).

In contrast look at what happens if the initialization for HBF are the data set points and the weights used are the ones learned by RBF (i.e. $c = K^+\hat{Y}$):
In this case a more reasonable error curve is observed. The error is more in line with the performance RBF and is much lower than when the centers are random noise. This seems to be a difference with Neural Networks that seem to learn good functions as long as there is symmetry breaking in the initial weights (even when the weights are randomly chosen).

3.3 Comparing the neural networks and Hyper Basis Functions (HBF)

In this section we will compare the Neural Networks with the ReLu and sigmoid activation (as well as their fixed center counter parts) against Hyper Basis Functions with moving centers with the best initialization that I found to work best. For the HBF we used random data set points as initialization the weights from the RBF. For Neural Net ReLu and Sigmoid, we used the traditional \( w \sim N(0, \epsilon^2) \). As a base line we will be using PCA.

From the the above figure we can see that surprisingly PCA is the best model
for auto-encoding. This is surprising because the other methods employ non-linear representations for auto-encoding, thus, it would have been natural to expect them to do well.

Also to my surprise, the HBF model and the RBF model performed quite similar, suggesting that training an HBF1 model is not as simple as the first task suggested.

The reason for this underperformance compared to the linear baseline is probably due to non-linear nature of the optimization. Thus, we hypothesize that one can possibly improve the performance of the non-linear models compared to PCA with enough patience in trying more different optimizers beyond SGD or Adagrad and many more combinations of hyper-parameters. Interesting optimizers to try would be Adam, RMSProp and Nesterov momentum, optimizers that proved useful in my studies.
Chapter 4

The Role of Structure in Learning Compositional Functions

4.1 Introduction

In this section we explore two main ideas. First, we show that there are statistical benefits of using networks that have similar (if not identical) compositional structure as the function they are approximating. We explore this task with standard Neural Networks and Hierarchical Hyper Basis functions (HBFs). Second, we explore the novel task of training Hierarchical Hyper Basis functions (HBFs). To the best of our knowledge, the optimization problem of Hierarchical HBFs has never been explored before. We show that training Hierarchical HBFs is hard and give a precise theoretical explanation for this phenomena. However, we also provide plenty of ideas and further research directions to tackle this new challenge.

4.2 Experimental setup

We want to compare how a shallow networks compares to deeper network and how depth plays a role in achieving good predictions. To make sure that the depth is the factor improving the model’s accuracy its important to have the same number of parameters in each network we compare. For example, comparing the accuracy
of a network with 1 layers and 10 units against a 2 layered network with 5 units in each layer is a fair comparison. When appropriate, the number of units is counts the numbers of parameters.

4.3 Learning a 1 dimensional compositional function

In this section we investigate the role of depth in learning the compositional function $f(x) = 2(2 \cos^2(x) - 1)^2 - 1$. Consider a visual representation of the function:

From this visualization it seems plausible that a model encoding the structure of the function to be learned as prior knowledge could significantly improve the performance of a network. Fortunately, it is known [7] that encoding such a prior guarantees the existence of good approximation function and we’d like to investigate the difficulty of finding good models.

4.3.1 Hierarchical neural networks

The following figures were borrowed from Hrushikesh et al [7].
This experiment portrays how the increase in depth in the model indeed increases the accuracy in the model when the complexity of the model in maintained fixed.

4.3.2 Hierarchical Hyper Basis Function networks

In this section we show the only positive result of training hierarchical HBFs. Consider the following figure:

This figure shows how hierarchical models, regardless of the activation function
can benefit from compositionally if the prior knowledge is known.

Furthermore, note that the error obtained by a hierarchical HBF is similar the hierarchical Neural Network, however, the number of centers need to achieve similar accuracy is much smaller for the HBF. This observation yields an insight that hierarchical HBF could potentially be excellent function approximations for compositional functions. This hypothesis is based on this empirical results coupled by the fact that Gaussian Activations encode a smoothness prior (from a regularization perspective) and could potentially be the reason for good predictions.

The error of the RBF was omitted, however, it was larger than HBF1 and HBF2.

4.3.3 Tricks for training the hierarchical HBF for \( f(x) = 2(2\cos^2(x) - 1)^2 - 1 \)

For the successful training of the HBF the following training conditions were used:

1. Good pre-trained initialization using the RBF. The final layer was equal to the least-squares solution to the kernelized RBF and the first layer were set to the subsampled data set points. Without this initialization, we experienced that training would get stuck since the beginning and not even improve in any sensible way.

2. The standard deviation of the first layer was in the same scale as the standard deviations that resulted in good models from the RBF. If this condition is not true, then the first layer provided no signal for learning because \( \|x - w^{(0)}\| \) would be at wrong scale and therefore the Gaussian activation would either be too small or too large all the time.

3. The standard deviation of the hidden layers were initialized to have a values in the range 1.5 to 3 and the initial filters where sampled from \( N(0, \epsilon) \). The scale \( \epsilon \) was chosen to be a value close to the activation activity of the previous layer to motivate \( \|x - w^{(0)}\| \approx 0 \) and propagate signal. Since the activation of the RBF is in the range \([0,1]\) its only necessary (at least initially) to have these conditions.
to promote learning. Large initial standard deviations and filters close to the scale of the activation of previous layers motivates learning initially.

4. Using an optimizer that adapts to each parameter individually and that is not very sensitive in the magnitude of the update proved important. Thus, using Adam, RMSProp or AdaGrad optimizer usually proved helpful for learning. For these experiments the Adam optimizer was used because it combined the advantages of the RMSProp optimizer and AdaGrad optimizer. Using an algorithm that was aggressive in the update (like momentum) prove a bad idea because it would usually change the (trainable) standard deviation of the Gaussian activation too much and learning would completely halt.

5. Using multiple trainable standard deviations for each RBF rather than a single shared one was also important in this experiment. What this means is that for a single layer of RBFs, it was useful to allow each RBF to have a different trainable standard deviation rather than all the RBF have the same parameter being shared (and trained).

![Errors vs centers](image)

The above figure shows how having multiple standard deviations clearly helped for the case of a singled layered HBF1. Fortunately, this was also helpful for the hierarchical HBF.
4.3.4 Discussion

It will be important to study if these heuristics generalize to other networks and data sets. From our experiments they did not easily work well on other data sets.

4.4 Learning a 2 dimensional compositional function

In this section we will investigate the role of depth in learning the compositional function $f_{task2}(x)$. This compositional function is defined by the following structure:

4.4.1 Neural Network training

In this section we show how a depth 2 Neural Network is able to beat a shallow network at approximating $f_{task2}(x)$:
4.5 Learning MNIST with Hierarchical Models

In this section we compare the reconstruction performance on MNIST of a single layer network and a two layered network.

The experiment was taken place so that the number of parameters in both networks was equal. Furthermore, about 60 random searches [3] over a chosen range of hyper parameters was done to find good settings of hyper parameters.
The singled layered network was trained with standard momentum while the two layered network was trained with RMSProp. There was a separate experiment were the two layered network was trained with batch normalization. The difference in reconstruction performance with batch normalization wasn’t significant so that result was omitted.

This experiment provides evidence real images even if they are simple might be composed of parts, similar to how compositional functions are composed. Since the number of units and parameters was kept fix and a good search over hyper parameters was done with the current optimization tools, it is reasonable to conclude that it was the role of depth that made the improved in performance.

4.6 Learning binary tree function with CNNs

In this section we will investigate the role of structure in learning the compositional function $f_{BT_{4D}}(x)$ with the shape of a binary tree. The goal will be to show that a Neural Network with the same structure as the function being learned can approximate it better than a shallow Network. Again we will keep the number of parameters constant to make sure the compositional structure of the network is what plays the main role in the improved performance. The function to be learned has the following structure

$$f_{BT_{4D}}(x_1, x_2, x_3, x_4) = h_2(h_{11}(x_1, x_2), h_{12}(x_3, x_4))$$

and can be visualized as a binary tree:

![Binary Tree Diagram](image)

The closed form equation for the specific function we will learned is:
\[ f_{BT_{4D}}(x_1, x_2, x_3, x_4) = k_3(k_1(x_1 + 2x_2)^2 + k_2(3x_3 + 4x_4)^4)^{0.5} \]

for this experiment the constant were:

\[ f_{BT_{4D}}(x_1, x_2, x_3, x_4) = \frac{1}{2} \left( \frac{1}{20}(x_1 + 2x_2)^2 + \frac{1}{10}(3x_3 + 4x_4)^4 \right)^{0.5} \]

4.6.1 Network structures

We will compare a shallow network (single layers) against a convolutional network with a binary tree structure.

Note however that each node in the binary tree will contain as many channels as there are convolutional filters.

4.6.2 Result of learning 4D binary tree function

In this section we present the comparison of training a shallow network against a convolutional binary tree network. Consider the following figure that shows the comparison between the shallow network and the binary tree network:

From these results we can appreciate that when the number of units is small, then having prior knowledge of the structure of the network gives the model an advantage when approximating the compositional function over the shallow model that does not encode that prior.
4.6.3 Discussion for 4D function

The experiments were done with the best of our by choosing hyper parameters manually. I’ve usually experienced more robust results when a large sample space of hyper parameters is searched automatically via random search [3].

Also notice that the results by Poggio et al. [7] is about the existence of an optimal solution. These results were obtained via optimization, which is almost surely suboptimal. Thus, having a large hyper parameter search with all known optimizers today could prove important to make results more robust.

Also, it will be interesting to try functions that are exactly the same across a single layer (including constants). This should make the gap bigger between binary tree Neural Net vs shallow Neural Net.

4.7 Learning an 8D binary tree function

In this section we will investigate the role of structure in learning the compositional function $f_{BT_{8D}}(x)$ with the shape of a binary tree. The goal will be to show that a Neural Network with the same structure as the function being learned can approximate it better than a shallow Network. Again we will keep the number of parameters constant to make sure the compositional structure of the network is what plays the
main role in the improved statistical performance. The function to be learned has
the following structure

\[ f(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = h_3(h_{21}(h_{11}(x_1, x_2), h_{12}(x_3, x_4)), h_{22}(h_{11}(x_5, x_6), h_{12}(x_7, x_8))) \]

and can be visualized as a binary tree:

```
      3
     / \   /
    /   /  /
   /   /   /
  /   /     /
 /   /       /
/   /         /
1   2   3   4
```

The closed form equation for the specific function we will learned is:

\[ f(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = h_3(h_{21}(h_{11}(x_1, x_2), h_{12}(x_3, x_4)), h_{22}(h_{11}(x_5, x_6), h_{12}(x_7, x_8))) \]

where each compositional function is as follows:

\[
\begin{align*}
  h_{11}(x_1, x_2) &= h_{13}(x_1, x_2) = \frac{1}{20} (x_1 + 2x_2)^4 \\
  h_{12}(x_1, x_2) &= h_{14}(x_1, x_2) = \frac{1}{2} (x_1 + 2x_2)^3 \\
  h_{21}(x_1, x_2) &= h_{22}(x_1, x_2) = \frac{1}{50} (5x_1 + 6x_2)^3 \\
  h_3(x_1, x_2) &= h_{22}(x_1, x_2) = \frac{1}{50} \left( x_1 + \frac{1}{100}x_2 + 1 \right)^{0.5}
\end{align*}
\]

4.7.1 Network structures

We will compare a shallow network (singled layers) against a convolutional network
with a binary tree structure:

Note however that each node in the binary tree will contain as many channels as
there are convolutional filters.
4.7.2 Results of learning 8D binary tree function

In this section we present the comparison of training a shallow network and a 8D binary tree convolutional network. Consider the following figure that shows the comparison between the shallow network and the binary tree network:

From these results we can appreciate that when the number of units is small, then having prior knowledge of the structure of the network gives the model an advantage when approximating the compositional function over the shallow model that does not encode that prior.
4.7.3 Discussion for 8D function

The experiments were done with the best of our by choosing hyper parameters manually. I’ve usually experienced more robust results when a large sample space of hyper parameters is searched automatically via random search [3].

Also notice that the results by Poggio et al. [7] is about the existence of an optimal solution. These results were obtained via optimization, which is almost surely suboptimal. Thus, having a large hyper parameter search with all known optimizers today could prove important to make results more robust.

Also, it will be interesting to try functions that are exactly the same across a single layer (including constants). This should make the gap bigger between binary tree Neural Net vs shallow Neural Net.
Chapter 5

Barrier to Training Hierarchical HBFs

5.1 The trade-off that results in permanent vanishing gradients

Let $V$ denote the loss function. Recall the back propagation equation for computing

$$
\delta_j^{(l)} = \frac{\partial V(w)}{\partial s_j^{(l)}};
$$

$$
\delta_j^{(l)} = \sum_{k'=1}^{D^{(l+1)}} \frac{\partial V(w)}{\partial s_{k'}^{(l+1)}} \frac{\partial a^{(l)}}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial s_j^{(l)}}
$$

Notice that the modified back propagation for a hierarchical HBF is as follow:

$$
\delta_j^{(l)} = \sum_{k'=1}^{D^{(l+1)}} \delta_{k'}^{(l+1)} W_{j,k'} \theta'(s_j^{(l)}) = \sum_{k'=1}^{D^{(l+1)}} \delta_{k'}^{(l+1)} [2\beta(w_{i,j}^{(l+1)} - a^{(l)})] \theta'(s_j^{(l)})
$$

Notice that $s_j^{(l)} = -\beta \|w_j^{(l+1)} - a^{(l)}\|^2$. Therefore, for the signal of back propagation to flow back through the network, we need two things: first, that the euclidian distance is small enough such that the Gaussian is not vanishingly small. Second, we need $\|w_j^{(l+1)} - a^{(l)}\|$ sufficiently large such that the difference between the current activation and the previous filter is not the zero vector i.e. we don’t want $w_j^{(l+1)} - a^{(l)}$ to be zero (or it to be really small). However, when $s_j^{(l)}$ increases, then the Gaussian activation will exponentially decay. This means that we need $w_j^{(l+1)} - a^{(l)}$ to be both small and large. These conditions are contradictory and makes hierarchical HBFs extremely
hard to train.

However, since the Gaussian decays exponentially faster than \(w_j^{(l+1)} - a^{(l)}\), one would expect that focusing on having \(\|w_j^{(l+1)} - a^{(l)}\|\) small enough should suffice for training. It happens that this sweet spot where its sufficiently large and small for training is extremely difficult to obtain. The only experiment that resulted in a successful actually required expensive automatic hyper parameter tuning, essentially brute force (after a human had manually found a good range to do the hyper parameter search). These are the reasons that proved training Hierarchical HBFs challenging.

5.1.1 Unstable pre-synaptic activation

One of the most challenging parts of training a Hierarchical HBF is getting the standard deviation right and having it stay in a regime that promotes the RBF activation throughout training (and not only at initialization). As training progresses there was evidence that \(s_j^{(l)} = -\beta^{(l)} \|w_j^{(l+1)} - a^{(l)}\|^2\) was rather unstable. The main ways to temporarily alleviate this was by having small learning rates or learning algorithms that didn’t use the size of the gradients to much to perform the updates. This would avoid \(\beta^{(l)}\) or \(w_{i,j}^{(l+1)}\) to shift to much. However, it was difficult to attain a point were the loss function decreased enough and the change in \(s_j^{(l)}\) was more stable. We expect that stabilizing this quantity should positively affect learning. Therefore, applied properly, batch normalization has the potential to stabilize this issue and in addition remove the covariate shift if that is the issue. Furthermore, since batch normalization is a technique that work during training, it could be a promising technique to apply correctly.
Chapter 6

Applying Batch Normalization to Hierarchical HBFs

In this section we suggest how to deal with unstable pre-synaptic activation using batch normalization for Hierarchical HBFs and explain the motivations for each suggestion. Most of the discussion will be under the framework that removing internal covariate shift (or as we dub it, stabilizing a distribution) aids in training. Our goal will be to seek this property under the assumption that it does indeed aid training. This assumption is based on its recent practical successes in training Neural Networks.

6.1 Background

Recall that batch normalization standardizes activations using mini-batch statistics with the goal of stabilizing distributions throughout training (or as the original paper dubs it "removing the internal covariate shift"). Let $h_i$ denote the inner product before the batch normalization:

$$h_i = \langle W_i, x \rangle = \sum_d W_{d,i} x_d$$

Furthermore, recall the batch normalization transform:
\[
\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}
\]

\[BN_{\gamma, \beta}(x_i) = \gamma_i \hat{x}_i + \beta_i\]

Iofee and Szegedy [6] recommend to apply the \(BN_{\gamma, \beta}\) transform on the inner products \(h_i\) before the non-linearity. To see why this is suggestion is sensible recall the definition of an inner product (for finite vectors):

\[
h_i = \langle W_i, x \rangle = \sum_d W_{d,i} x_d
\]

according to Iofee and Szegedy \(h_i\) is likely to (approximately) be a Gaussian Distribution [6] [5]. Intuitively, this is should be true because a sum of random variables (under certain conditions) tends to be Gaussian according to the Central Limit Theorem (CLT). Therefore applying batch normalization as follow:

\[
\hat{h}_i = BN_{1,0}(h_i) = BN_{0,1}(\langle W_i, x \rangle)
\]

should approximately result in a unit Gaussian \(N(0, 1)\) at every point during training. Therefore, the Neural Network does not need to re-adjust what it has learned due to the changes in distribution because \(\hat{h}_i\) is more stable. The reason is that it approximately stabilizes the distribution of inner products throughout learning by making them approximately unit Normal. In addition, this phenomena should be stronger the more units a network has. Note however, that Iofee and Szegedy suggest to include trainable \(\gamma, \beta\) because allows the net to re-scale and shift this Gaussian if necessary. It also retains the Network’s expressive capacity.

### 6.2 Naive batch normalization on hierarchical HBF

With the goal of stabilizing the distribution being fed into the exponential function one could naively batch normalize the squared euclidian distance. Doing:
\[ \hat{h}_i = BN_{\gamma, \beta}(||w_{i,j}^{(l+1)} - a^{(l)}||^2) \]

probably eliminates the covariate shift because it can be expressed as a sum of random variable, however, it introduces another significant problem. By normalizing the squared euclidian distance around zero we introduced the problem that the input to the Gaussian activation could potentially be negative when in fact it always expects distances with are positive. There always exists at least one negative value (unless the values) and thus, if there exists \( \hat{h}_i < 0 \) then \(-\hat{h}_i > 0\) which implies that the input to the exponential function could be positive. By allowing the exponential function to receive positive numbers the activation does not remain Gaussian and in fact becomes exponential and unbounded. This causes the training to explode and makes the model useless. This explains the extremely unstable training when this method was applied.

### 6.3 Suggestion 1: applying batch normalization to euclidian distance before squaring

One can apply batch normalization to the euclidian distance before its squared and then square it:

\[ Y_j^{(l)} = [BN_{\gamma, \beta}(||w_j^{(l+1)} - a^{(l)}||)]^2 \]

This solves the issue of the non-negativity as input to the Gaussian activation because the square is always non-negative.

However, does it remove the covariate shift? Recall the definition of euclidian distance \( ||w_j^{(l+1)} - a^{(l)}|| = \sqrt{\sum(w_{i,j}^{(l+1)} - a_i^{(l)})^2} \). The inner part of the summation approaches a normal with mean \( n\mu \) and standard deviation \( \sqrt{n}\sigma \). Batch normalizing it results in the following:
\[ BN_{\gamma,\beta}(\|w_j^{(l+1)} - a^{(l)}\|)^2 = BN_{\gamma,\beta} \left( \sqrt{\sum (w_{i,j}^{(l+1)} - a_i^{(l)})^2} \right) \]

Unfortunately, even though \( \sum (w_{i,j}^{(l+1)} - a_i^{(l)})^2 \) approaches a Gaussian, the square root of it is not guaranteed to remain Gaussian and thus, batch normalizing does not guarantee to make it unit Normal nor remove the covariate. Thus, it would be surprising to see batch normalization help training if applied in this way.

### 6.4 Suggestion 2: apply batch normalization to squared Euclidian distance and square it again

We propose to instead batch normalize as follow:

\[ Y_j^{(l)} = \left[ BN_{\gamma,\beta} \left( \|w_j^{(l+1)} - a^{(l)}\| \right) \right]^2 \]

This suggestion seems more sensible because \( \|w_j^{(l+1)} - a^{(l)}\| = \sum (w_{i,j}^{(l+1)} - a_i^{(l)})^2 \) approaches a Gaussian \( N(n\mu, \sqrt{n}\sigma) \). Therefore, Normalizing it does have the effect of removing the covariate shift when \( \gamma = 1, \beta = 0 \) and hence resulting in \( Y_j \sim [N(0,1)]^2 \) at every step of training. First note that an activation \( a_j \) would be computed as follow \( a_j = exp(-Y_j) \) and since \( Y_j > 0 \) removes the issue of exploding activations. However, it would result in a distribution with the following mean and variance:

\[ E[Y_j] = \sigma^2 + \mu^2 = 1 \]

\[ Var[Y_j] = E[Y_j^4] - \sigma^4 - \mu^4 - 2\sigma^2\mu^2 = E[Y_j^4] - 1 \]

Unfortunately the fact that the expectation is 1 means that value of the activation could potentially be really small. However, this can be potentially solved by introducing a parameter \( a \) that acts as a scale:

\[ \frac{1}{a^2} BN(\|w_{i,j}^{(l+1)} - a^{(l)}\|)^2 \]
choosing \(a\) well could drive the expected value of \(\frac{1}{a^2}Y\) closer to zero and hence promote learning where \(exp(-x^2)\) function is largest. Note that if one choses to train \(a\) then one has to divide by \(a^2\) to maintain the overall input to the activation negative.

The importance of training \(a\) is open but if it gets to small too quickly it could stop training prematurely. If one decides to fix it a good value could be chosen by cross validation. However, choosing any value that makes it closer to 1 could and could be chosen randomly.

Alternatively one can also introduce a shift parameter \(b = Var[Y] - \epsilon\). However, it one requires calculating \(E[Y^4]\). Note that the shift parameter is ok to introduce as long as training it does not make it to negative.

### 6.5 Suggestions to fix the optimization procedure for hierarchical HBFs

In this section we will try to address the permanent vanishing gradient problem (i.e. the problem of having requiring \(s^{(l)}_j = -\beta \|w^{(l+1)}_j - a^{(l)}\|^2\) to be both simultaneously large and small).

### 6.6 Normalize filters and activations of the Gaussian

An idea to solve this issue could be to normalize the filters and activations such that their norm is equal to 1. Doing this could potentially remove the term \(w^{(l+1)}_j - a^{(l)}\) from the back propagation equation and remove the issue entirely. Recall the pre synaptic activation in this case:

\[
s^{(l)}_j = -\beta \|a^{(l-1)} - w^{(l)}_{i,j}\|^2 = \beta \langle a^{(l-1)}, w^{(l)}_{i,j} \rangle - \beta^{(l)}
\]

Taking the derivative of the above would mean we are left with a factor \(w^{(l)}_j\) instead of the pesky difference \(w^{(l+1)}_j - a^{(l)}\). This would result in the new back propagation equation:
\[
\delta_j^{(l)} = \sum_{k'=1}^{D^{(l+1)}} \delta_{k'}^{(l+1)} \left[ 2\beta \hat{w}_{i,j}^{(l+1)} \right] \theta'(s_j^{(l)})
\]

instead of

\[
\delta_j^{(l)} = \sum_{k'=1}^{D^{(l+1)}} \delta_{k'}^{(l+1)} \left[ 2\beta (w_j^{(l+1)} - a^{(l)}) \right] \theta'(s_j^{(l)})
\]

This seems promising because it removes the requirement of having \(w_j^{(l+1)} - a^{(l)}\) be both large and small for successful training.

However, it might require projected gradients and it could potentially be important to make SGD aware of the normalization for the optimization to work.

Though the main advantage again is that this would allow us to control the Euclidian distance being close to zero without compromising the remaining terms in the back propagation equation.

What would remain to do is how to eliminate the covariate shift in this case.

### 6.7 Use the Laplacian function instead of Gaussian

In this section we suggest to use the smooth approximation to the triangle function as an activation function instead of the Gaussian. Recall the Laplacian function:

\[
\theta(a - w) = e^{-\beta (|a^{(l)} - w_j^{(l)}|)}
\]

Similar to the previous section, this should remove the vanishing gradient problem described in section 5.1. The reason is nearly identical, however one has \(\pm w_{i,j}^{(l+1)}\) instead of \(w_j^{(l+1)} - a^{(l)}\) (i.e. the sub-gradient of \(|a^{(l)} - w_j^{(l)}|\)). This results in the following back propagation equation:

\[
\delta_j^{(l)} = \sum_{k'=1}^{D^{(l+1)}} \delta_{k'}^{(l+1)} \hat{W}_{j,k'} \theta'(s_j^{(l)}) = \sum_{k'=1}^{D^{(l+1)}} \delta_{k'}^{(l+1)} 2\beta \left[ \pm w_{i,j}^{(l+1)} \right] \theta'(s_j^{(l)})
\]

which clearly removes the permanent vanishing gradient problem because the euclidian distance is allowed to approach zero without making \(\delta_j^{(l)}\) approach zero. This
is because its not a function not a function of \( w_j^{(l+1)} - a^{(l)} \).

### 6.7.1 Suggestion on applying Batch Norm to Laplacian pre-activation

In the previous section we suggested to use the Laplacian to solve the permanent vanishing gradient problem. In this section we extend this ideas and suggest how to use Batch Normalization to remove the internal covariate shift problem.

The first ideas is to (naively) apply batch normalization to \( |w_j^{(l+1)} - a^{(l)}| \), however, it would results in exploding gradient because of the exponential activation function. However a quick fix is to apply batch normalization to \( |w_{i,j}^{(l+1)} - a^{(l)}| \) and then taking the absolute value again to avoid the batch normalization from giving us negative values. This could potentially results in the following equation:

\[
Y = |BN(|w_{i,j}^{(l+1)} - a^{(l)}|)| \sim |N(0, 1)|
\]

If this happens then it means that the input to the exponential is \( Y_i \), which is a stable distribution because it receives the half normal distribution (i.e. \(|N(0, 1)|\)) at every iteration of training. The only issue is that the half normal has first and second moments as follows:

\[
E[X] = \frac{\sigma \sqrt{2}}{\sqrt{\pi}} \approx 0.797988
\]
\[
Var[X] = \sigma^2 (1 - \frac{2}{\pi}) \approx 0.363602
\]

This means the normalized pre-activations are not close to zero. If this is true it means that even if it is stable, it will not fire the activation enough to propagate signal through the network.

There are two potential way to fix this by introducing additional parameters \( a, b \) as follow:

\[
\frac{1}{a^2} (|N(0, 1)| - b)
\]
To require that the input to the exponential be positive one can choose $b = 0$ and only keep the scale $a$. The scale can be chosen to be fixed and greater than 1 to aid learning.

An initial idea is to have the constraint that $b \geq \sigma^2(1 - \frac{2}{\pi})$, however, it does not keep the non-negativity invariant that we desire if we want to avoid exploding gradients. It’s still unclear how to introduce a shift parameter and keep this non-negativity constraint.
Chapter 7

Further Work

7.1 Memory minimization

One hypothesis that we want to explore further is memory minimization with hierarchical HBFs. For data that is compositional in nature it is interesting to explore how hierarchical models, such as the one in hierarchical HBFs, can take advantage of this composition and use reusable parts to minimize the memory needed to learn and possibly, have the same or better statistical performance. Furthermore, it will be interesting to explore what type of templates (centers) hierarchical HBFs learn and compare it with the centers learned in single layered RBF and DCNs.

7.2 Removing internal covariate shift for hierarchical HBFs

It would be interesting to see how the ideas of adapting batch-normalization explained in chapter 6 would work on Hierarchical HBFs.
7.3 Removing the trade off for signal propagation in hierarchical HBFs

It would be interesting to see how the suggestions in chapter 6 would work. Remove $a - w$ by changing the network pre activations to absolute distance rather than Euclidian distance.

7.4 Using HBFs to beat other machine learning algorithms on function approximation

The Gaussian function is derived from a smoothness constraint in the regularization. Thus, it sensible to hypothesize that hierarchical HBFs should be much better at function approximation. There is already evidence provided in this thesis because the one positive result of Hierarchical HBFs achieves the same accuracy as Neural Networks but with many less centers.

It would also be interesting the explore the idea of having all the hidden layers be ReLu units and take advantage of the compositionally with depth this way but in the final layer allow the function approximation layer be an HBF. This model should be much easier to train and might perform better at function approximation.

7.5 Regularization

Possibly having regularization on weights could help training. The idea is that if the weights on hidden layers (except the first layer) are kept close to zero (or close to the activation of the previous layer), then the Euclidian distance should be small. This should avoid the vanishing problem of Gaussian functions by having their input small. However, this idea needs to be made more precise.

Starting places can be some modification of KL divergence or L2 norm as regularizers.
Chapter 8

Appendix

8.1 Update equations using back-propagation for layered HBF network

It is worth including the update equations in vectorized for because they were a bit of work to vectorize and make them work (and are good for reference and understanding the code). It is worth reviewing the equations for Neural Networks for back-propagation to understand this section best. Most of the difference between NN and HBFs are because we consider the pre-activation:

\[-\beta \|a^{(l-1)} - w_{i,j}^{(l)}\|^2 = 2\beta \langle a^{(l-1)}, w_{i,j}^{(l)} \rangle - \beta (\|a^{(l-1)}\|^2 + \|w_{i,j}^{(l)}\|^2)\]

compared to just a dot product:

\[\langle a^{(l)}, w_{i,j}^{(l)} \rangle + b_j^{(l)}\]

This will mean that we will often see differences (like \(a^{(l-1)} - w_{i,j}^{(l)}\) or \(w_{i,j}^{(l)} - a^{(l-1)}\)) in the derivative equations.
8.2 Equations for derivatives

$$\frac{\partial V(w)}{\partial w_{i,j}^{(l)}} = \frac{\partial V(w)}{\partial s_j^{(l)}} \frac{\partial s_j^{(l)}}{\partial w_{i,j}^{(l)}} = \delta_j^{(l)} a_{i,j}^{(l)} = \delta_j^{(l)} 2\beta(a_{i}^{(l-1)} - w_{i,j}^{(l)})$$

Usually the above can be easily computed with an outer product for a neural network, however, the presence of two indices for $\tilde{a}_{i,j}^{(l)}$ complicates that. Thus the derivative that we want is (for some mini batch):

$$\frac{\partial J(X_{1:M})}{\partial w_{i,j}^{(l)}} = \sum_{m=1}^{M} \frac{\partial V(x(m), y(m))}{\partial w_{i,j}^{(l)}} = \sum_{m=1}^{M} \delta(m)_j^{(l)} 2\beta(a_{i}^{(l-1)} - w_{i,j}^{(l)})$$

$$\frac{\partial J(X_{1:M})}{\partial w_{i,j}^{(l)}} = 2\beta(\sum_{m=1}^{M} \delta(m)_j^{(l)} a_{i}^{(l-1)} - \sum_{m=1}^{M} \delta(m)_j^{(l)} w_{i,j}^{(l)})$$

where the first term can be easily computed with an outer product and the second term can also be computed by defining the tensor:

$$T_{i,j,m}^{(l)} = \delta(m)_j^{(l)} w_{i,j}^{(l)}$$

which can be show to be computed via:

$$T^{(l)} = bsxfun(@times, W^{(l)}, \Delta^{(l)})$$

which gets us the vectorized equation:

$$\frac{\partial J(X_{1:M})}{\partial W^{(l)}} = 2\beta(A^{(l-1)}\Delta^{(l)} - \text{sum}(T^{(l)}, 3))$$

8.3 Computing delta

Recall the definition of delta:

$$\frac{\partial V(w)}{\partial s_j^{(l)}} = \delta_j^{(l)}$$

using the multi-variable chain rule its not hard to see:
\[ \delta_j^{(l)} = \sum_{k'=1}^{D^{(l+1)}} \frac{\partial V(w)}{\partial s_{k'}^{(l+1)}} \frac{\partial a_j^{(l)}}{\partial s_{k'}^{(l)}} = \sum_{k'=1}^{D^{(l+1)}} \delta_{k'}^{(l+1)} \tilde{W}_{j,k'}^{(l+1)} \theta'(s_{j}^{(l)}) \]

Unfortunately, because \( \tilde{W}_{j,k'}^{(l+1)} = 2\beta(w_{i,j}^{(l+1)} - a_i^{(l)}) \) we couldn’t quite see how to fully re-use the original back-propagation equations, though, they are re-used to compute half of the above equation. First notice:

\[ \delta_j^{(l)} = 2\beta \left( \sum_{k'=1}^{D^{(l+1)}} \delta_{k'}^{(l+1)} w_{i,j}^{(l+1)} - \sum_{k'=1}^{D^{(l+1)}} a_i^{(l)} \theta'(s_{j}^{(l)}) \right) \]

thus, the first term can be simply computed using the traditional back-propagation equation (and be stacked with the other delta’s that resulted from the mini-batch). This leads to the following vectorized code:

\[ \Delta^{(l)} = 2\beta \theta'(S^{(l)}) \bullet (\delta^{(l+1)} W^{(l+1)T} - A^{(l)} \bullet sum(\delta^{l+1}, 2)) \]

where the last \( \bullet \) denotes the hadamard product (element-wise product). Notice that the last hadamard product has an implicit repmat to match the dimensions of \( A^{(l)} \). Also, notice that the upper case letter of the variable means that we have stacked each vector corresponding to its mini batch index in the current SGD iteration. For example, the first row of \( A^{(l)} \) is the activation resulting from the first data set in the batch. Thus, \( A \in \mathbb{R}^{M \times D^{(l)}} \).

### 8.4 GPUs

Notice that because of the above vectorization, one can now add very few lines of code to matlab code and be able to use GPUs. I did use GPUs and observed about 6-fold speed up. One can see the changes to the code to use GPU in my github.

### 8.5 Software

The software that we used was a small library that we implemented in MATLAB that can be found in our github page [8]. The library was fully tested, specially for
numerical gradients. The HBF was implemented in MATLAB. Once TensorFlow [1] was released we migrated to TensorFlow and implemented our Neural Networks and Hierarchical Hyper Basis Function with Tensorflow. All the code can be found in our public github site [9].
Bibliography


